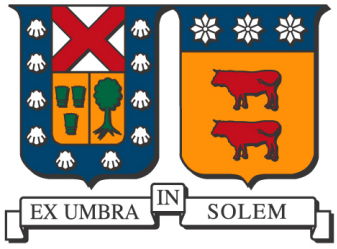
Investigación de Operaciones
ILN250

Tarea Computacional I

Integrantes:
José Antonio Catalán Fuentes
Daniel Alexander Loaiza Trujiillo



Universidad Técnica Federico Santa María
2021 - 1

Contents

Departamento de industrias

Junio 2021

1 Resumen

El objetivo de esta investigación es la de emplear distintos métodos para solucionar el **Problema de mínimos cuadrados**. Los métodos de descenso a utilizar son **gradiente descendente y Newton**, calculando el tamaño de paso con **exact line search**. Antes de la implementación del método, fue necesario calcular los respectivos pesos w_i asociados al valor de cada punto experimental con tal de que fuesen convenientes para la solución. Finalmente, se realizó una comparación de tiempos de cómputo y las iteraciones necesarias para la convergencia del resultado.

2 Introducción

El Problema de mínimos cuadrados, es un problema de modelamiento en donde se pretende encontrar los valores del vector z que minimice la suma de los errores cuadrados, considerando el peso respectivo a cada punto.

$$\min_z \sum_{i \in I} w_i e(x_i, z)^2$$

Donde $e(x_i, z)$ corresponde a la aproximación de curva lineal o cuadrática y el vector z, corresponde a los valores reales de cada función respectivas.

- $e(x_i, z)^2 = (y_i - g(x_i))^2$
- $e(x_i, z)^2 = (y_i - h(x_i))^2$
- $g(x_i) = a + bx$
- $h(x_i) = c + dx + ex^2$

Los respectivos valores experimentales para (x_i, y_i), son entregados como parte del problema.

3 Valores de pesos

Dentro de los modelos estadísticos para encontrar parámetros a partir de regresiones tenemos uno de los más utilizados como el de mínimos cuadrados ordinarios, sin embargo, el rango de problemas a resolver con dicho método está acotado a la condición de homocedasticidad, el cual nos indica que la varianza del error asociado a las variables es constante para todas las observaciones.

Lo cual no es el caso de los datos experimentales entregados, ya que poseen una desviación estándar variable según el dato medido, y por tanto se dice que presenta heterocedasticidad; la que se resuelve aplicando mínimos cuadrados ponderados, con pesos respectivos w_i, estos pesos nos ayudan a minimizar el error y encontrar modelos más eficaces.

Como la varianza de los errores se puede conocer dado la regresión respectiva y los datos experimentales, diremos que los pesos tendrán la siguiente forma:

$$w_i = \frac{1}{\sigma_i^2}$$

Donde se define el valor de un peso con relación inversamente proporcional a la varianza de los datos, es decir, se le asignará un valor de peso mayor a valores más cercanos a la regresión dada, o en su defecto, valores que tengan una menor distancia respecto al ajuste correspondiente, lo cual nos dará un modelo con mayor exactitud y menor sensibilidad a datos extremos.

Para efectos del problema se considerará desviación estándar o error en cada y_i como:

$$\sigma_I = y_i - \overline{y}$$

Con y_i como el valor experimental y $\overline{y}$ el valor calculado según la regresión correspondiente (con pesos unitarios).

Obteniendo los siguientes valores:

Ajuste Lineal		**Ajuste Cuadrático**	
Error	Pesos (w_i)	Error	Pesos (w_i)
1,29	0,60	0,10	94,53
0,32	9,48	0,63	2,52
0,44	5,08	0,12	70,00
0,49	4,21	0,27	14,02
0,43	5,35	0,51	3,92
0,36	7,92	0,67	2,21
0,77	1,69	0,26	15,22
0,80	1,57	0,14	52,67
0,14	50,53	0,62	2,64
0,93	1,16	0,12	65,26
0,91	1,21	0,10	92,18
0,70	2,07	0,06	283,17
0,89	1,28	0,23	19,06
0,86	1,36	0,35	8,24
0,60	2,81	0,28	12,42
0,28	12,79	0,21	22,09
0,26	14,94	0,03	1035,03
0,22	20,35	0,35	8,19
1,16	0,74	0,20	25,28
1,88	0,28	0,47	4,44

4 Análisis de convexidad

Realizando el reemplazo respectivo para la función $g(x_i)$, podemos obtener que la función a minimizar es la siguiente.

$$\min_{a,b} \sum_{i \in I} w_i (y_i - a - b x_i)^2$$

La hessiana de la función respectiva quedaría de la siguiente manera:

$$\sum_{i \in I} \begin{bmatrix} 2w_i & 2w_i x_i \\ 2w_i x_i & 2w_i x_i^2 \end{bmatrix}$$

De donde podemos obtener sus sub. determinantes

$$\nabla_1 = 2w_i > 0$$

$$\nabla_2 = 4w_i^2 x_i^2 - 4wi_i^2 x_i^2 = 0$$

Considerando los valores de los sub determinantes, podemos concluir que la hessiana es semi-definida positiva, por lo cual la función asociada es **convexa**.
De la misma manera, podemos analizar la gráfica, en donde podemos ver que es **convexa** en todo su dominio.

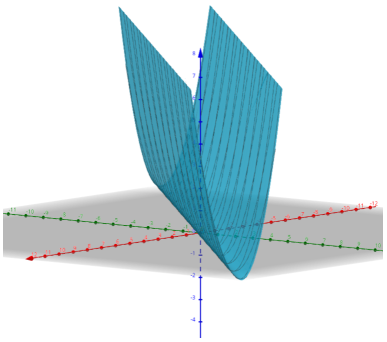


Figure 1: Gráfica función asociada a g()

Para la función **h(c,d,e)**, tenemos la siguiente Hessiana asociada:

$$\sum_{i \in I} \begin{bmatrix} 2w_i & 2w_i x_i & 2w_i x_i^2 \\ 2w_i x_i & 2w_i x_i^2 & 2w_i x_i^3 \\ 2w_i x_i^2 & 2w_i x_i^3 & 2w_i x_i^4 \end{bmatrix}$$

El resultado de los sub-determinantes es:

$$\nabla_1 = 2w_i > 0$$

$$\nabla_2 = 4w_i^2 x_i^2 - 4wi_i^2 x_i^2 = 0$$

$$\nabla_2 > 0$$

Considerando estos valores, podemos concluir que la matriz hessiana es semi-definida positiva, por lo que su función asociada es **convexa**. Cabe destacar que ∇_2, fue calculado computacionalmente, el código asociado a este cálculo se muestra a continuación:

```
if __name__ =="__main__":
    wi, xi, yi, c, d, e, t, n, i =sp.symbols('wi xi yi c d e t n i')
    wi, xi, yi, a, b, c, t, n,i =sp.symbols('wi xi yi a b c t n i')

    f_ab =sp.Sum(sp.Indexed('wi',i) *(sp.Indexed('yi',i) -a -
                                            b*sp.Indexed('xi',i))**2, (i,0,19) )
    f_cde =sp.Sum(sp.Indexed('wi',i) *( sp.Indexed('yi',i) -c -d*sp.Indexed('xi',i) -
                                            e*(sp.Indexed('xi',i))**2 )**2,
                                            (i,0,19) )
    f = sp.lambdify([xi, yi, wi], f_cde)
    #f = sp.lambdify([xi, yi, wi], f_ab)

    hess =calculate_hessian(f_cde, symbols=[c,d,e])
    sp.init_printing(use_latex=True)
    sp.pprint(sp.Matrix(hess))
    det =sp.Matrix(hess).det()
    det =sp.lambdify([xi, yi, wi], det)

    sp.pprint(det(data_xi,data_yi,data_wi))
```

4.1 Análisis de gráficos

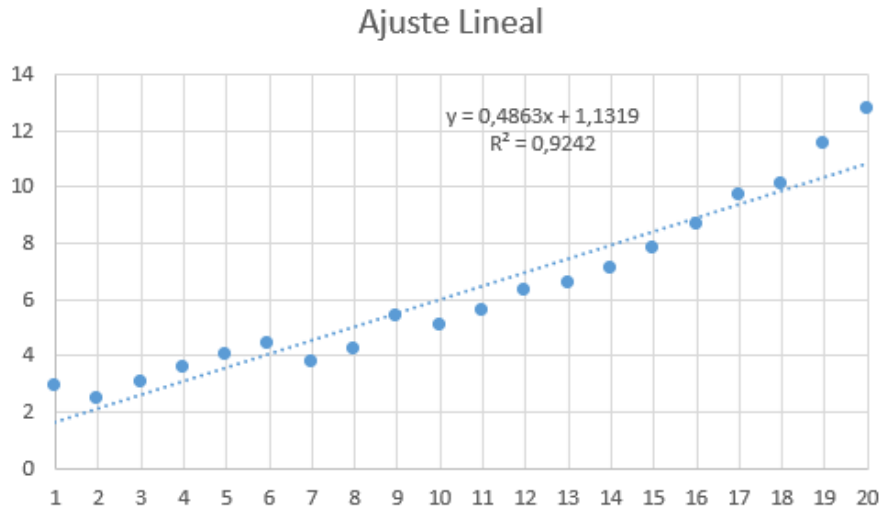


Figure 2: Ajuste $g(x)$

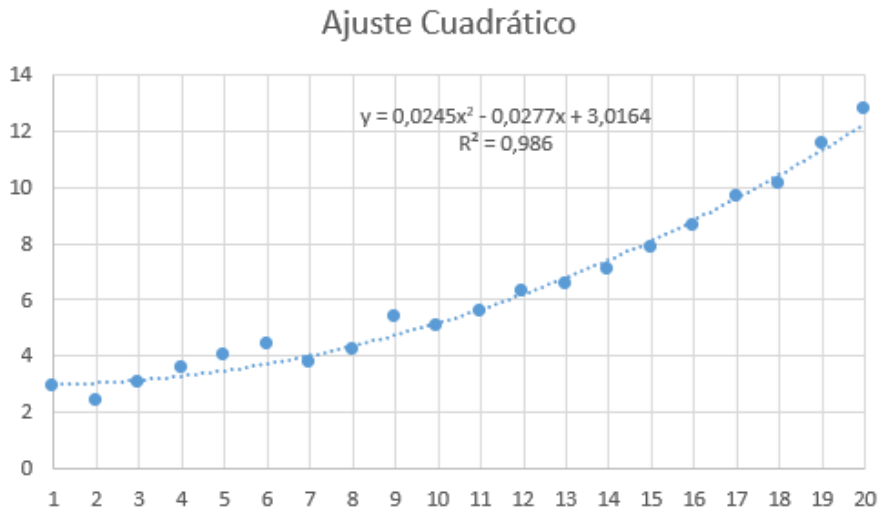


Figure 3: Ajuste $h(x)$

El coeficiente de determinación R^2 dado por Excel nos dice que el ajuste cuadrático es más cercano a 1, y por lo tanto este modelo se ajusta más a los datos dados, esto también se puede comprobar visualmente a través de las gráficas.

5 Conclusión

En conclusión, fue posible implementar los algoritmos previamente mencionados, en el lenguaje **python3**, permitiéndonos ver el comportamiento del algoritmo para distintos puntos de partida.
Para la obtención del promedio de **CPU TIME**, se realizaron 10 pruebas idénticas, permitiendo obtener un promedio de estas. Cabe destacar que las pruebas del método de newton, fueron realizadas en una laptop **thinkpad x1, CPU: Intel i7-9750H (12)**
Para pesos neutros, los resultados se muestran a continuación.

z_0	Newton		Gradiente		z^*	z^*_{grad}
(0, 0)	0.331233 [s]	2 iter.	0.9071693[s]	2 iter.	(1.131908, 0.486261)	(1.125261, 0.483405)
(100, 100)	0.373057 [s]	2 iter.	2.917601 [s]	7 iter.	(1.131908, 0.486261)	(1.131908, 0.486260
(1000, 1000)	0.360407 [s]	2 iter.	3.340392 [s]	9 iter.	(1.131908, 0.486261)	(1.131907, 0.486260)

Table 1: Tiempos de cómputo y número de iteraciones para la función $g(a, b)$, con pesos unitarios.

z_0	Newton		Gradiente		z^*_{newton}	z^*_{grad}
(0, 0, 0)	0.445687 [s]	2 iter.	2206,266 [s]	7493 iter.	3.016370, -0.027683, 0.024474)	(3.016367, -0.027682, 0.024473)
(100, 100, 100)	0.476976 [s]	2 iter.	$\approx$ inf [s]	n iter.	(3.016370, -0.027683, 0.024474)	(a*,b*)
(1000, 1000, 1000)	0.488793 [s]	2 iter.	$\approx$ inf [s]	n iter.	(3.016370, -0.027683, 0.024474)	(a*,b*)

Table 2: Tiempos de cómputo y número de iteraciones para la función $h(c, d, e)$, con pesos unitarios.

Luego de realizar el cálculo de los pesos respectivos, se procedió a iterar nuevamente, obteniéndose los siguientes resultados. (El código asociado a la obtención de los pesos puede ser encontrada en la sección Anexos - cálculo de pesos).

z_0	Newton		Gradiente		z^*	z^*_{grad}
(0, 0)	0.594854 [s]	2 iter.	1.186341 [s]	2 iter.	(1.246769, 0.477082)	(1.239597, 0.474337)
(100, 100)	0.576273 [s]	2 iter.	3.394552 [s]	7 iter.	(1.246769, 0.477082)	(1.246769, 0.477081)
(1000, 1000)	0.556556 [s]	2 iter.	4.0981032 [s]	9 iter.	(1.246769, 0.477082)	(1.246769, 0.477081)

Table 3: Tiempos de cómputo y número de iteraciones para la función $g(a, b)$ con pesos definidos.

z_0	Newton		Gradiente		z^*_{newton}	z^*_{grad}
(0, 0, 0)	0.950682 [s]	2 iter.	971.3318 [s]	2795 iter.	(2.922830, -0.027857, 0.024930)	(2.922829, -0.027857, 0.024930)
(100, 100, 100)	0.961562 [s]	2 iter.	$\approx$ inf [s]	n iter.	(2.922830, -0.027857, 0.024930)	(a*,b*)
(1000, 1000, 1000)	0.999041 [s]	2 iter.	$\approx$ inf [s]	n iter.	(2.922830, -0.027857, 0.024930)	(a*,b*)

Table 4: Tiempos de cómputo y número de iteraciones para la función $h(c, d, e)$ con pesos definidos.

Como se puede apreciar, el aumento de tiempo de cómputo para el método de **Newton** es mínimo, esto puede deberse a optimizaciones de la librería **sympy**, al ver los pesos unitarios. Además de esto, se logra apreciar una leve variación el el Z^*_{newton}, cercana al $\approx 9\%$, para el caso lineal, y $\approx 2\%$ para el caso cuadrático, considerando el parámetro R, descrito en la sección "Análisis de gráficos", podemos corroborar que la aproximación con menor variación es la cuadrática, ya que es la que más se ajusta a los datos entregados.

Además, es posible observar un mejor desempeño utilizando el método de **newton**. Esto puede deberse a la simplicidad de las funciones trabajadas, ya que las operaciones sobre el **Hessiano** de las mismas, no tienen una complejidad algorítmica elevada. También hay que considerar que no se realizó ningún tipo de optimización en la implementación de los algoritmos mencionados, por ejemplo, evitar cómputos repetitivos. Esto pudo haber afectado de alguna manera el **CPU time** obtenido, aún así, no es razón suficiente para haber obtenido una variación considerable para demostrar una notoria mejoría en el desempeño de los algoritmos.

6 Anexo

Los códigos asociados a las implementaciones se pueden encontrar en el siguiente enlace https://github.com/jcatala/gio_tarea1_2021-1

6.1 cálculo de pesos

```
#!/usr/bin/venv python3


import sympy as sp
import numpy as np
import time

data ={1   :2.910348,2  :2.429320,3  :3.034274,4  :3.564630,5  :3.995516,6  :4.404796,7
                                          :3.766659,8  :4.225016,9  :5.367586,10 :
                                          5.065610,11 :5.572047,12 :6.271426,13 :
                                          6.567767,14 :7.082164,15 :7.829593,16 :
                                          8.632454,17 :9.657083,18 :10.106290,19 :
                                          11.533470,20 :12.736908
}

data_xi =[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
data_yi =[2.910348, 2.429320,3.034274,
3.564630,3.995516,4.404796,3.766659,4.225016,5.367586,5.065610,5.572047,6.271426,
6.567767,7.082164,7.829593,8.632454,9.657083,10.106290,11.533470,12.736908]
data_wi =[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]

def pesos_g():
    # considerando ajuste de curva con pesos unitarios
    x,y =sp.symbols('x y')
    f =1.131908 +0.486261*x
    f_l =sp.lambdify([x], f)
    weights =[]
    for k in range(20):
        exp     = data_yi[k]
        aprox   = f_l(data_xi[k])
        err =abs(exp -aprox)
        wi =1/(err**2)
        weights.append(wi)
        #sp.pprint("Experimental: %f, teorico: %f, err: %f, wi: %f" % (exp, aprox,
                                                  err, wi))
    sp.pprint("Weights for g(a,b):")
    sp.pprint(weights)

def pesos_h():
    x,y,z =sp.symbols('x y z')
    f =3.016370 +-0.027683*x+ 0.024474*(x**2)
    sp.pprint(f)
    f_l =sp.lambdify([x], f)
    weights =[]
    for k in range(20):
        exp     = data_yi[k]
        aprox   = f_l(data_xi[k])
        err     = abs(exp -aprox)
        wi =1/(err**2)
        weights.append(wi)
```

```
        #sp.pprint("Experimental: %f, teorico: %f, err: %f, wi: %f" % (exp, aprox,
                                                    err, wi))
    sp.pprint("Weights for h(c,d,e):")
    sp.pprint(weights)

if __name__ =="__main__":
    pesos_g()
    pesos_h()
```